Lecture 16: BLOCKING / NON-BLOCKING ASSIGNMENTS (PART 1)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Procedural Assignment

- Procedural assignment statements can be used to update variables of types "reg", "integer", "real" or "time".
- The value assigned to a variable remains unchanged until another procedural assignment statement assigns a new value to the variable.
  - This is different from continuous assignment (using "assign") that results in the expression on the RHS to continuously drive the "net" type variable on the left.
- Two types of procedural assignment statements:

  a) Blocking (denoted by "=")

  b) Non-blocking (denoted by "<=")

- The left-hand side of a procedural assignment statement can be one of:
  a) A register type variable (“reg”, “integer”, “real”, or “time”)
  b) A bit select of these variables (e.g. sum[15])
  c) A part select of these variables (e.g. IR[31:26])
  d) A concatenation of any of the above
- The right-hand side can be any expression consisting of “net” and “register” type variables that evaluates to a value.
- Procedural assignment statements can only appear within procedural blocks (“initial” or “always”).

(i) Blocking Assignment

- General syntax:

```
variable_name = [delay_or_event_control] expression;
```

- The “=“ operator is used to specify blocking assignment.
- Blocking assignment statements are executed in the order they are specified in a procedural block.
  – The target of an assignments gets updated before the next sequential statement in the procedural block is executed.
  – They do not block execution of statements in other procedural blocks.
- This is the recommended style for modeling combinational logic.

- Blocking assignments can also generate sequential circuit elements during synthesis (e.g. incomplete specification in multi-way branching with "case").
- An example of blocking assignment:

```
integer  a, b, c;
initial
  begin
    a = 10; b = 20; c = 15;
    a = b + c;
    b = a + 5;
    c = a – b;
  end
```

- Initially, a=10, b=20, c=15
- a becomes 35
- b becomes 40
- c becomes -5


```
Module blocking_example;
  reg X, Y, Z;
  reg [31:0] A, B;    integer sum;

  initial
    begin
      X = 1'b0;  Y = 1'b0;  Z = 1'b1;    // At time = 0
      sum = 1;                           // At time = 0
      A = 31'b0;  B = 31'hababababab;    // At time = 0
      #5  A[5] = 1'b1;                   // At time = 5
      #10 B[31:29] = {X, Y, Z};          // At time = 15
      sum = sum + 5;                     // At time = 15
    end
endmodule
```

Simulation of an Example

```
module blocking_assgn;
integer a, b, c, d;
always @ (*)
  repeat (4)
  begin
    #5 a = b + c;
    #5 d = a - 3;
    #5 b = d + 10;
    #5 c = c + 1;
  end
```

```
initial
  begin
    $monitor ($time, "a=%4d, b=%4d,
        c=%4d, d=%4d", a, b, c, d);
    a = 30; b = 20; c = 15;  d = 5;
    #100 $finish;
  end
endmodule
```



```
 0 a=  30, b=  20, c=  15, d=   5
 5 a=  35, b=  20, c=  15, d=   5
10 a=  35, b=  20, c=  15, d=  32
15 a=  35, b=  42, c=  15, d=  32
20 a=  35, b=  42, c=  16, d=  32
25 a=  58, b=  42, c=  16, d=  32
30 a=  58, b=  42, c=  16, d=  55
35 a=  58, b=  65, c=  16, d=  55
40 a=  58, b=  65, c=  17, d=  55
45 a=  82, b=  65, c=  17, d=  55
50 a=  82, b=  65, c=  17, d=  79
55 a=  82, b=  89, c=  17, d=  79
60 a=  82, b=  89, c=  18, d=  79
65 a= 107, b=  89, c=  18, d=  79
70 a= 107, b=  89, c=  18, d= 104
75 a= 107, b= 114, c=  18, d= 104
80 a= 107, b= 114, c=  19, d= 104
```

Simulation Results

```
Initially:
  a=30, b=20, c=15, d=5
always @ (*)
  repeat (4)
  begin
    #5 a = b + c;
    #5 d = a - 3;
    #5 b = d + 10;
    #5 c = c + 1;
  end
```

(ii) Non-Blocking Assignment

- General syntax:

```
variable_name <= [delay_or_event_control] expression;
```

- The "<=" operator is used to specify non-blocking assignment.
- Non-blocking assignment statements allow scheduling of assignments without blocking execution of statements that follow within the procedural block.
  - The assignment to the target gets scheduled for the end of the simulation cycle (at the end of the procedural block).
  - Statements subsequent to the instruction under consideration are not blocked by the assignment.
  - Allows concurrent procedural assignment, suitable for sequential logic.

- This is the recommended style for modeling sequential logic.
  - Several "reg" type variables can be assigned synchronously, under the control of a common clock.

```
integer  a, b, c;
initial
  begin
    a = 10; b = 20; c = 15;
  end
initial
  begin
    a <= #5 b + c;
    b <= #5 a + 5;
    c <= #5 a - b;
  end
```

- Initially, a=10, b=20, c=15
- a becomes 35 at time = 5
- b becomes 15 at time = 5
- c becomes -10 at time = 5

All the right hand side expressions are evaluated together based on the previous values of "a", "b" and "c". They are assigned together at time 5.

```
always @(posedge clk)
  begin
    a <= b & c;
    b <= a ^ d;
    c <= a | b;
  end
```

Recommended style for modeling synchronous circuits, where assignments take place in synchronism with clock.

All assignments take place synchronously at the rising edge of the clock.

Swapping values of two variables "a" and "b"

```
always @(posedge clk)
  a = b;
always @(posedge clk)
  b = a;
```

- Either a=b will execute before b=a, or vice versa, depending on simulator implementation.
- Both registers will get the same value (either "a" or "b").
  - *Race condition*.

```
always @(posedge clk)
  a <= b;
always @(posedge clk)
  b <= a;
```

- Here the variables are correctly swapped.
- All RHS variables are read first, and assigned to LHS variables at the positive clock edge.

Trying to swap using blocking assignment

```
always @(posedge clk)
  begin
    a = b;
    b = a;
  end
```

- Both “a” and “b” will be getting the value previously stored in “b”.

```
always @(posedge clk)
  begin
    ta = a;
    tb = b;
    a = tb;
    b = ta;
  end
```

- Correct swapping will occur, but we need two temporary variables “ta” and “tb”

Simulation of an Example

```
module nonblocking_assgn;
integer a, b, c, d;
reg clock
always @ (posedge clock)
  begin
    a <= b + c;
    d <= a - 3;
    b <= d + 10;
    c <= c + 1;
  end
```

```
initial
  begin
    $monitor ($time, "a=%4d, b=%4d,
        c=%4d, d=%4d", a, b, c, d);
    a = 30; b = 20; c = 15;  d = 5;
    clock = 0;
    forever  #5 clock = ~clock;
  end
initial
    #100 $finish;
endmodule
```

Simulation Results

```
 0 a=  30, b=  20, c=  15, d=   5
 5 a=  35, b=  15, c=  16, d=  27
15 a=  31, b=  37, c=  17, d=  32
25 a=  54, b=  42, c=  18, d=  28
35 a=  60, b=  38, c=  19, d=  51
45 a=  57, b=  61, c=  20, d=  57
55 a=  81, b=  67, c=  21, d=  54
65 a=  88, b=  64, c=  22, d=  78
75 a=  86, b=  88, c=  23, d=  85
85 a= 111, b=  95, c=  24, d=  83
95 a= 119, b=  93, c=  25, d= 108
```

```
Initially:
  a=30, b=20, c=15, d=5
always @ (posedge clock)
  begin
    a <= b + c;
    d <= a - 3;
    b <= d + 10;
    c <= c + 1;
  end
```


Some Rules to be Followed

- It is recommended that blocking and non-blocking assignments *are not mixed* in the same "always" block.
  - Simulator may allow, but this is not good design practice.
- Verilog synthesizer ignores the delays specified in a procedural assignment statement (blocking or non-blocking).
  - May lead to functional mismatch between the design model and the synthesized netlist.
- A variable cannot appear as the target of both a blocking and a non-blocking assignment.
  - This is not permissible →

```
x = x + 5;
x <= y;
```

END OF LECTURE 16

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Lecture 17: BLOCKING / NON-BLOCKING ASSIGNMENTS (PART 2)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Introduction

- We shall be looking at some examples of modeling using blocking and non-blocking assignments.
- Objective is to get a feel of the type of assignment statement to use for some particular scenario.
- Avoid some of the "not-so-good" design practices in modeling.


```
// 8-to-1 multiplexer: behavioral description
module  mux_8to1 (in, sel, out);
  input [7:0] in;   input [2:0] sel;
  output reg out;
  always  @(*)
    begin
      case (sel)
        3'b000: out = in[0];
        3'b001: out = in[1];
        3'b010: out = in[2];
        3'b011: out = in[3];
        3'b100: out = in[4];
        3'b101: out = in[5];
        3'b110: out = in[6];
        3'b111: out = in[7];
        default: out = 1'bx;
      endcase
    end
endmodule
```

```
// Up-down counter (synchronous clear)
module  counter (mode, clr, ld, d_in, clk, count);
   input mode, clr, ld, clk;
   input [0:7] d_in;
   output reg [0:7]  count;

   always  @ (posedge  clk)
     if (ld)          count <= d_in;
     else if (clr)    count <= 0;
     else if (mode)   count <= count + 1;
     else             count <= count - 1;
endmodule
```


Make a design general for any number of bits.

Using the keyword "*parameter*".

- Parameter values are substituted before simulation or synthesis.

```
// Parameterized design:: an N-bit counter

module  counter (clear, clock, count);
    parameter  N = 7;
    input  clear, clock;
    output reg [0:N] count;

    always @(negedge  clock)
       if (clear)
          count <= 0;
       else
          count <= count + 1;
endmodule
```

```
// Using more than one clocks in a module
module  multiple_clk (clk1, clk2, a, b, c, f1, f2);
    input  clk1, clk2, a, b, c;
    output reg f1, f2;
    always @(posedge  clk1)
        f1 <= a & b;
    always @(negedge  clk2)
        f2 <= b ^ c;
endmodule
```



```
// Using multiple edges of the same clock
module multi_edge_clk (a, b, f, clk);
    input  a, b, clk;
    output reg f;  reg t;
    always @(posedge  clk)
        f  <=  t & b;
    always @(negedge  clk)
        t  <=  a | b;
endmodule
```

```
// Another example
module multi_edge_clk (a, b, c, d, f, clk);
    input  a, b, clk;
    output reg f;  reg t;
    always @(posedge  clk)
        c  <=  a + b;
    always @(negedge  clk)
        f  <=  c - d;
endmodule
```

- Two operations are carried out every clock cycle.
  - "c" is assigned at the rising edge.
  - "f" is assigned at the falling edge.
- It is assumed that addition or subtraction can be completed in half a clock cycle.


```
// A ring counter
module  ring_counter  (clk, init, count);
  input  clk, init;
  output reg [7:0]  count;
  always  @ (posedge clk)
    begin
       if (init)  count = 8'b10000000;
        else  begin
                  count    = count << 1;
                  count[0] = count[7];
               end
    end
endmodule
```

- This solution is wrong.
- count[7] will get overwritten in the first statement.
  - Rotation of the bits will not happen.

```
// A ring counter (Modified version 1)
module  ring_counter  (clk, init, count);
  input  clk, init;
  output reg [7:0]  count;
  always  @ (posedge clk)
    begin
       if (init)  count = 8'b10000000;
        else  begin
                 count    <= count << 1;
                 count[0] <= count[7];
               end
    end
endmodule
```

- This is the correct version.
- Since non-blocking assignments are used, rotation will take place correctly.


```
// A ring counter (Modified version 2)
module  ring_counter  (clk, init, count);
  input  clk, init;
  output reg [7:0]  count;
  always  @ (posedge clk)
    begin
       if (init)  count = 8'b10000000;
       else
          count = {count[6:0], count[7]};
    end
endmodule
```

- This is a correct way of modeling using blocking assignment.

END OF LECTURE 17

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Lecture 18: BLOCKING / NON-BLOCKING ASSIGNMENTS (PART 3)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Blocking vrs. Non-blocking Assignments

- We shall now illustrate some examples that show how the modeling style influences the simulator or synthesizer to capture the behavior of the modeled circuit.
  - Very important concept required to be clearly understood by the designer.
  - Even a slight error in modeling can result in a drastically different circuit.
- Highly recommended:
  - For any confusion, write a Verilog code, simulate it and analyze the output(s).

Example 1

```
begin
  a = #5 b;
  c = #5 a;
end
```

- The value of "b" will be assigned to "c" 10 time units after the "begin … end" block starts.

```
begin
  a <= #5 b;
  c <= #5 a;
end
```

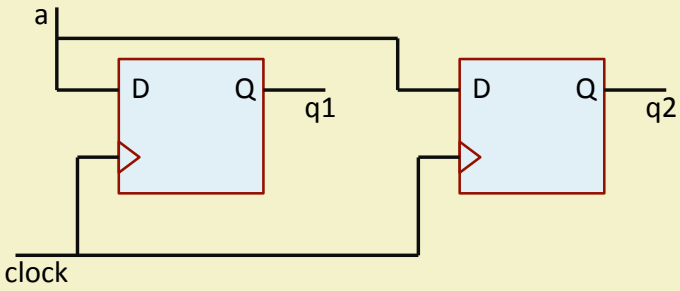
- "a" is scheduled to get the value of "b" 5 time units into the future.
- "c" is also scheduled to get the value of "a" 5 time units into the future.

Value of "c" will be different for the two cases

Example 2

```
always @(posedge clock)
  begin
    q1 = a;
    q2 = q1;
  end
```

Parallel Flip-flops

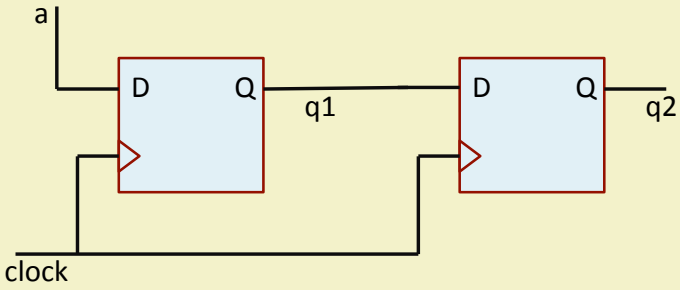


Example 3

```
always @(posedge clock)
  begin
    q2 = q1;
    q1 = a;
  end
```

```
always @(posedge clock)
  begin
    q1 <= a;
    q2 <= q1;
  end
```

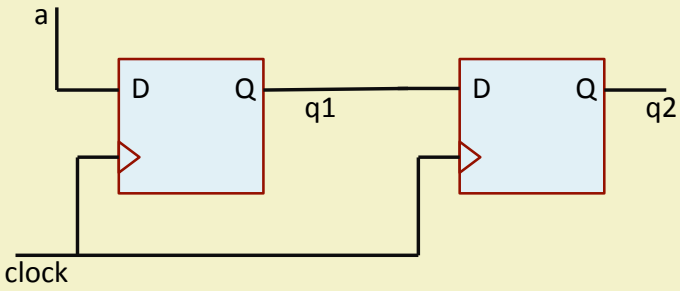
Shift Register



Example 4

```
always @(posedge clock)
    q2 <= q1;
always @(posedge clock)
    q1 <= a;
```

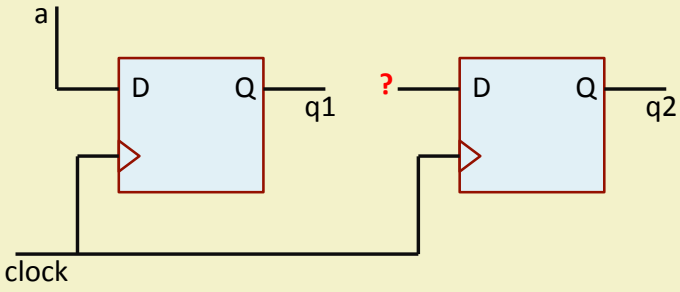
Shift Register



Example 5

```
always @(posedge clock)
    q1 = a;
always @(posedge clock)
    q2 = q1;
```

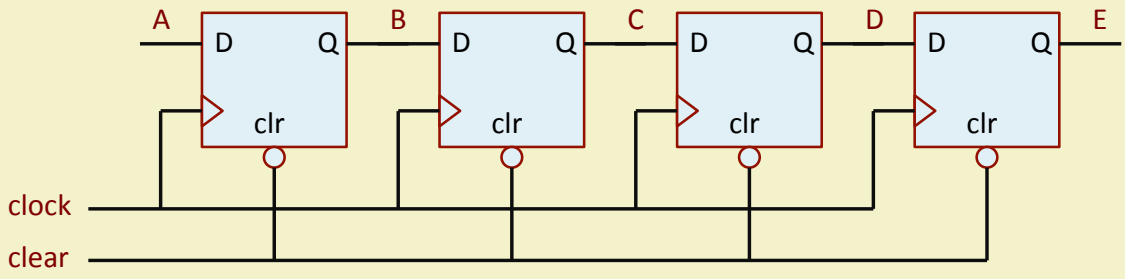
q2 will be indeterminate



Example 6

What circuit will the synthesis tool generate?

```
module shiftreg_4bit (clock, clear, A, E);
  input clock, clear, A;
  output reg E;
  reg B, C, D;
  always @(posedge clock or negedge clear)
    begin
      if (!clear) begin B=0; C=0; D=0; E=0; end
      else begin
             E = D;
             D = C;
             C = B;
             B = A;
           end
    end
endmodule
```

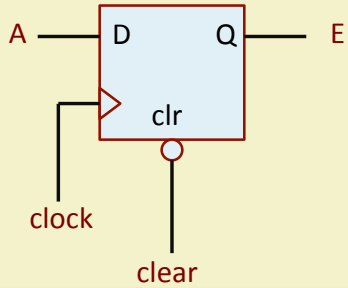
Example 6a

We just reverse the order of the procedural assignments.

What circuit will the synthesis tool generate?

```
module shiftreg_4bit (clock, clear, A, E);
  input clock, clear, A;
  output reg E;
  reg B, C, D;
  always @(posedge clock or negedge clear)
    begin
      if (!clear) begin B=0; C=0; D=0; E=0; end
      else begin
             B = A;
             C = B;
             D = C;
             E = D;
           end
    end
endmodule
```


- The effect of the assignment made by the first statement (B = A) is immediate.
- Thus, B changes, and the updated value is used in the second statement (C = B).
- The updated value of C is used in the third statement (D = C).
- The updated value of D is used in the fourth statement (E = D).
- The statements execute sequentially.
  - But at the same time step of the simulator.
  - The four statements are equivalent to a single statement that assigns A to E.



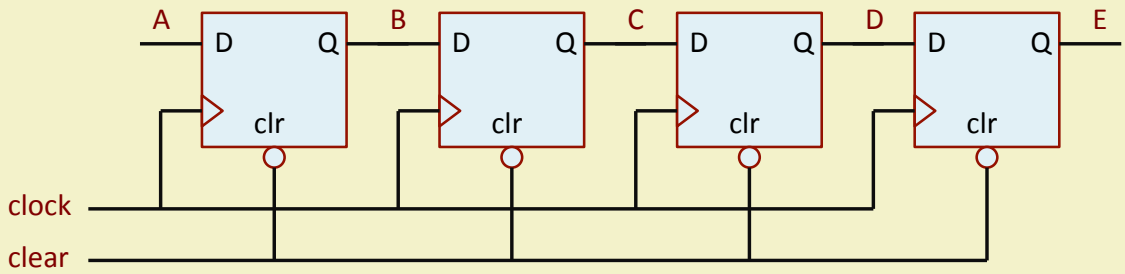
Example 6b

Recommended style for modeling sequential circuits.

- Statements can appear in any order.
- Chances of errors are less.

```
module shiftreg_4bit (clock, clear, A, E);
  input clock, clear, A;
  output reg E;
  reg B, C, D;
  always @(posedge clock or negedge clear)
    begin
      if (!clear) begin B<=0; C<=0; D<=0; E<=0; end
      else begin
             E <= D;
             D <= C;
             C <= B;
             B <= A;
           end
    end
endmodule
```

The right-hand side expressions are evaluated in parallel, so that order of the statements is not important.



END OF LECTURE 18

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

Lecture 19: BLOCKING / NON-BLOCKING ASSIGNMENTS (PART 4)

PROF. INDRANIL SENGUPTA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MIXING BLOCKING AND NON-BLOCKING ASSIGNMENTS IN A PROCEDURAL BLOCK

NOT RECOMMENDED

Basic Idea

- It is possible to combine both blocking and non-blocking assignments in the same procedural block (viz. “always”).
  - Simulator or synthesis tool supports this type of usage.
- However, interpretation of the circuit behavior under such mixed usage is not very straightforward.
  - Not recommended for designers.
  - We shall explain the semantics using two simple examples.
- Such mixing should be avoided in a design.

Example 1

```
always @(*)
  begin
    x = 10;
    x = 20;
    y = x;
    #10;
  end
```

```
always @(*)
  begin
    x = 10;
    x = 20;
    y <= x;
    #10;
  end
```

- Same result will be shown for both the versions:
  - "x" will be assigned 10 and then 20, both at time 0.
  - The value of "x" at time 0 will be assigned to "y".

x = 20, y = 20

Example 2

```
always @(*)
  begin
    x = 10;
    y = x;
    x = 20;
    #10;
  end
```

```
always @(*)
  begin
    x = 10;
    y <= x;
    x = 20;
    #10;
  end
```

- Same result will be shown for both the versions:
  - "y" will be assigned 10 at time 0.
  - The final value of "x" will be 20.

x = 20, y = 10

Generate Blocks

- "*generate*" statements allow Verilog code to be generated dynamically before the simulation or synthesis begins.
  - Very convenient to create parameterized module descriptions.
  - Example: N-bit ripple carry adder for arbitrary value of N.
- Requires the keywords "*generate*" and "*endgenerate*".
- Generate instantiations can be carried out for various Verilog blocks:
  - Modules, user-defined primitives, gates, continuous assignments, "initial" and "always" blocks, etc.
  - Generated instances have unique identifier names and can be referenced hierarchically.

- Special "*genvar*" variables:
  - The keyword "*genvar*" can be used to declare variables that are used only in the evaluation of generate block.
  - These variables do not exist during simulation or synthesis.
  - The value of a "*genvar*" can be defined only in a generate loop.
  - Every generate loop is assigned a name, so that variables inside the generate loop can be referenced hierarchically.

Example 1

```
module xor_bitwise (f, a, b);
  parameter N = 16;
  input [N-1:0] a, b;
  output [N-1:0] f;
  genvar p;

  generate for (p=0; p<N; p=p+1)
      begin xorlp
        xor XG (f[p], a[p], b[p]);
      end
  endgenerate
endmodule
```

```
module generate_test;

reg [15:0] x, y;
wire [15:0] out;

xor_bitwise G (.f(out),.a(x),.b(y));

initial
  begin
    $monitor ("x: %b, y: %b,
            Out: %b", x, y, out);
    x = 16'haaaa;  y = 16'h00ff;
    #10 x = 16'h0f0f;  y = 16'h3333;
    #20 $finish;
  end
endmodule
```


Simulation Results

```
x: 1010101010101010, y: 0000000011111111, Out: 1010101001010101
x: 0000111100001111, y: 0011001100110011, Out: 0011110000111100
```

- In the bitwise xor example, the name “xorlp” was given to the generate loop.
- The relative hierarchical names of the xor gates will be:

  `xorlp[0].XG, xorlp[1].XG, …, xorlp[15].XG`

Example 2: Design of N-bit Ripple Carry Adder

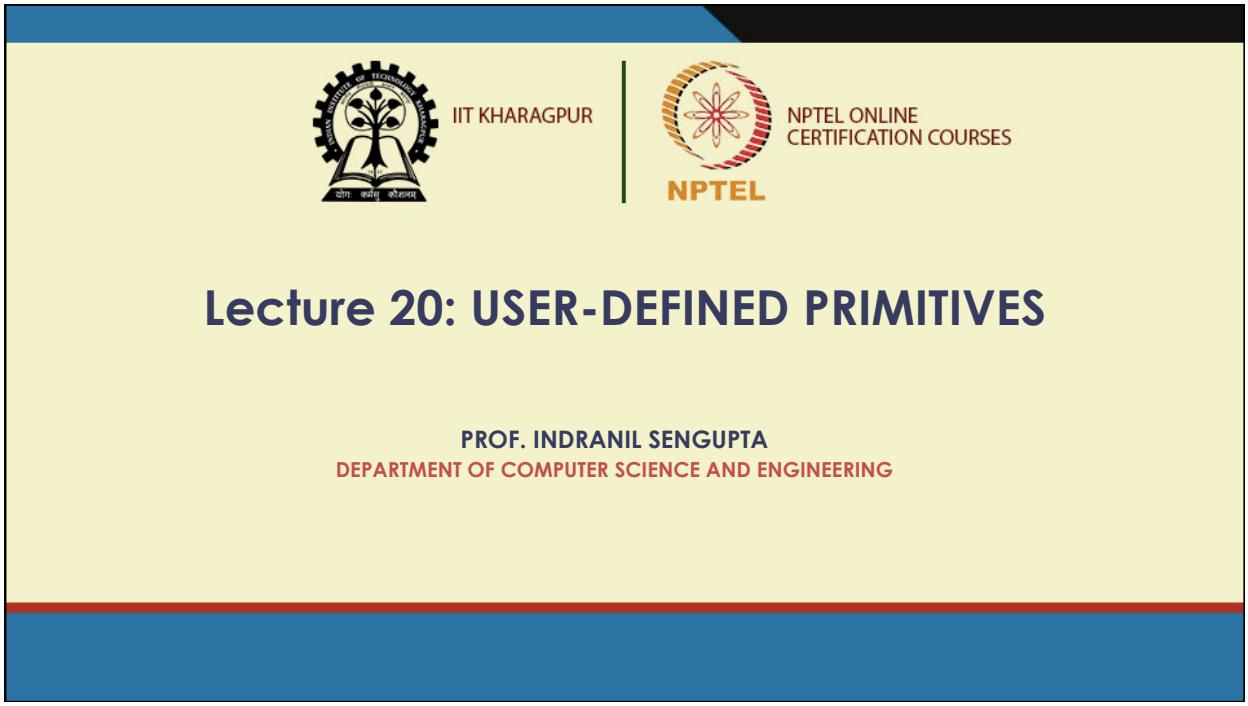
```
// Structural gate-level description of a full adder
module full_adder (a, b, c, sum, cout);
  input a, b, c;
  output, sum, cout;
  wire t1, t2, t3;
  xor G1 (t1, a, b), G2 (sum, t1, c);
  and G3 (t2, a, b), G4 (t3, t1, c);
  or G5 (cout, t2, t3);
endmodule
```

How to use "generate" to dynamically create N copies of full adder, and connect them to make a N-bit ripple-carry adder?


```
module RCA (carry_out, sum, a, b, carry_in);
  parameter N = 8;
  input [N-1:0] a, b;    input carry_in;
  output [N-1:0] sum,    output carry_out;
  wire [N:0] carry;  // carry[N] is carry out
  assign carry[0]  = carry_in;
  assign carry_out = carry[N];
  genvar i;
  generate for (i=0; i<N; i++)
    begin  fa_loop
      wire t1, t2, t3;
      xor G1 (t1, a[i], b[i]), G2 (sum[i], t1, carry[i]);
      and G3 (t2, a[i], b[i]), G4 (t3, t1, carry[i]);
      or G5 (carry[i+1], t2, t3);
    end
  endgenerate
endmodule
```

- Some of the relative hierarchical instance names that are generated are:
  - fa_loop[0].G1, fa_loop[1].G1, fa_loop[7].G1, etc.
- Some of the nets (“wires”) that are generated are:
  - fa_loop[0].t1, fa_loop[1]. T2, fa_loop[0].t3, etc.

END OF LECTURE 19



User Defined Primitives (UDP)

- They are used to define custom Verilog primitives by the use of lookup tables.
- They can specify:
  - Truth table for combinational functions.
  - State table for sequential functions.
  - Don't care, rising and falling edges, etc. can be specified.
- For combinational functions, truth table entries are specified as:

  <input1> <input2> … <inputN> : <output>;
- For sequential functions, state table entries are specified as:

  <input1> <input2> … <inputN> : <present_state> : <next_state>

Some Rules for using UDPs

- The input terminals to a UDP can only be scalar variables.
  - Multiple input terminals can be used.
  - The input terminals are declared as "*input*".
  - Input entries in the table must be in the same order as the "*input*" terminal list.
- Only one scalar output terminal must be used.
  - The output terminal must appear in the beginning of the terminal list.
  - For combinational UDPs, the output terminal is declared as "*output*".
  - For sequential UDPs, the output terminal is declared as "*reg*".
- For sequential UDPs, the state can be initialized with an "*initial*" statement.
  - This is optional.

Some Guidelines

- User defined primitives (UDPs) model functionality only.
  - They do not model timing or process technology.
- A functional block can be modeled as a UDP only if it has exactly one output.
  - If a block has more than one outputs, it has to be modeled as a module.
  - As an alternative, multiple UDPs can be used, one per output.
- Inside the simulator, a UDP is typically implemented as a lookup table in memory.
- The UDP state tables should be specified as completely as possible.
  - For unspecified cases, the output is set to "x".

MODELING COMBINATIONAL CIRCUITS


```
// Full adder sum generation using UDP
primitive udp_sum (sum, a, b, c);
  input a, b, c;
  output sum;
  table
  // a  b  c    sum
     0  0  0  :  0;
     0  0  1  :  1;
     0  1  0  :  1;
     0  1  1  :  0;
     1  0  0  :  1;
     1  0  1  :  0;
     1  1  0  :  0;
     1  1  1  :  1;
  endtable
endprimitive
```

The truth table is specified for all input combinations.

We can also specify don't care input combinations as "?".

```
// Full adder carry generation
primitive udp_cy (cout, a, b, c);
  input a, b, c;
  output cout;
  table
  // a  b  c    cout
     0  0  0  :  0;
     0  0  1  :  0;
     0  1  0  :  0;
     0  1  1  :  1;
     1  0  0  :  0;
     1  0  1  :  1;
     1  1  0  :  1;
     1  1  1  :  1;
  endtable
endprimitive
```

```
// Full adder carry generation
// Using don't care ("?")
primitive udp_cy (cout, a, b, c);
  input a, b, c;
  output cout;
  table
  // a  b  c    cout
     0  0  ?  :  0;
     0  ?  0  :  0;
     ?  0  0  :  0;
     1  1  ?  :  1;
     1  ?  1  :  1;
     ?  1  1  :  1;
  endtable
endprimitive
```



```
// Instantiating UDP's
// A full adder description

module full_adder (sum, cout, a, b, c);
  input  a, b, c;
  output sum, cout;

  udp_sum  SUM  (sum, a, b, c);
  udp_cy  CARRY (cout, a, b, c);
endmodule
```

UDPs can be instantiated just like any other Verilog module.

```
// A 4-input AND function
primitive udp_and4 (f, a, b, c, d);
  input a, b, c, d;
  output f;
  table
  // a  b  c  d     f
     0  ?  ?  ?  :  0;
     ?  0  ?  ?  :  0;
     ?  ?  0  ?  :  0;
     ?  ?  ?  0  :  0;
     1  1  1  1  :  1;
  endtable
endprimitive
```

```
// A 4-input OR function
primitive udp_or4 (f, a, b, c, d);
  input a, b, c, d;
  output f;
  table
  // a  b  c  d     f
     1  ?  ?  ?  :  1;
     ?  1  ?  ?  :  1;
     ?  ?  1  ?  :  1;
     ?  ?  ?  1  :  1;
     0  0  0  0  :  0;
  endtable
endprimitive
```



```
// A 4-to-1 multiplexer
primitive udp_mux41 (f, s0, s1, i0, i1, i2, i3);
  input  s0, s1, i0, i1, i2, i3;
  output f;
  table
  // s0 s1   i0 i1 i2 i3  :  f
     0  0    0  ?  ?  ?   :  0;
     0  0    1  ?  ?  ?   :  1;
     1  0    ?  0  ?  ?   :  0;
     1  0    ?  1  ?  ?   :  1;
     0  1    ?  ?  0  ?   :  0;
     0  1    ?  ?  1  ?   :  1;
     1  1    ?  ?  ?  0   :  0;
     1  1    ?  ?  ?  1   :  1;
  endtable
endprimitive
```

MODELING SEQUENTIAL CIRCUITS


```
// A level-sensitive D type latch

primitive Dlatch (q, d, clk, clr);
  input  d, clk, clr;
  output reg q;

  initial
    q = 0;   // This is optional

  table
  //  d  clk clr     q   q_new
      ?   ?   1   :  ?  :  0;    // latch is cleared
      0   1   0   :  ?  :  0;    // latch is reset
      1   1   0   :  ?  :  1;    // latch is set
      ?   0   0   :  ?  :  -;    // retains previous state
  endtable
endprimitive
```

```
// A T flip-flop

primitive TFF (q, clk, clr);
  input  clk, clr;
  output reg q;

  table
  //  clk   clr     q   q_new
      ?     1     :  ?  :  0;    // FF is cleared
      ?    (10)   :  ?  :  -;    // ignore –ve edge of "clr"
     (10)  0     :  1  :  0;    // FF toggles at –ve edge of "clk"
     (10)  0     :  0  :  1;    // - do -
     (0?)  0     :  ?  :  -;    // ignore +ve edge of "clk"
  endtable
endprimitive
```



```
// Constructing a 6-bit ripple counter using T flip-flops

module ripple_counter (count, clk, clr);
  input  clk, clr;
  output [5:0] count;

  TFF F0 (count[0], clk, clr);
  TFF F1 (count[1], count[0], clr);
  TFF F2 (count[2], count[1], clr);
  TFF F3 (count[3], count[2], clr);
  TFF F4 (count[4], count[3], clr);
  TFF F5 (count[5], count[4], clr);
endmodule
```

```
// A negative edge sensitive JK flip-flop
primitive JKFF (q, j, k, clk, clr);
  input  j, k, clk, clr;
  output reg q;
  table
  //  j    k    clk  clr     q   q_new
      ?    ?    ?    1    :  ?  :  0;    // clear
      ?    ?    ?   (10)  :  ?  :  -;    // ignore .. no change
      0    0   (10)  0    :  ?  :  -;    // no change
      0    1   (10)  0    :  ?  :  0;    // reset condition
      1    0   (10)  0    :  ?  :  1;    // set condition
      1    1   (10)  0    :  0  :  1;    // toggle condition
      1    1   (10)  0    :  1  :  0;    // toggle condition
      ?    ?   (01)  0    :  ?  :  -;    // no change
  endtable
endprimitive
```



```
// A positive edge sensitive SR flip-flop
Primitive SRFF (q, s, r, clk, clr);
  input  s, r, clk, clr;
  output reg q;
  table
  //  s    r    clk  clr     q   q_new
      ?    ?    ?    1    :  ?  :  0;    // clear
      ?    ?    ?   (10)  :  ?  :  -;    // ignore .. no change
      0    0   (01)  0    :  ?  :  -;    // no change
      0    1   (01)  0    :  ?  :  0;    // reset condition
      1    0   (01)  0    :  ?  :  1;    // set condition
      1    1   (01)  0    :  ?  :  x;    // invalid condition
      ?    ?   (10)  0    :  ?  :  -;    // ignore .. no change
  endtable
endprimitive
```

Some Rules to Follow

- The "?" symbol cannot be specified in an output field.
- The "-" symbol, indicating no change in the state value, can be specified only in an output field.
- The shortcut "r", indicating rising edge, can be used instead of (01).
- The shortcut "f", indicating falling edge, can be used instead of (10).
- The shortcut "*" indicates any value change in the signal.

END OF LECTURE 20